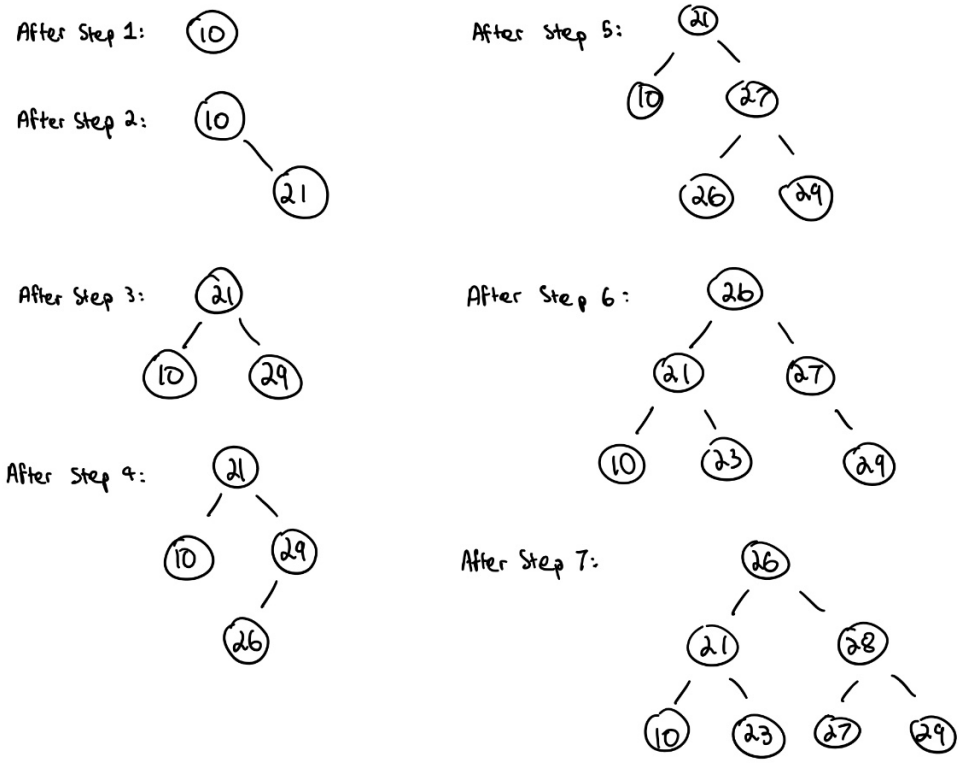
Task 6.3 To attain quadratic running time, we realize that our current recursive solution repeats recursive cases (it tries to figure out whether the same string is a hashtag multiple times). Hence, to fix this, we start at the end of the string and check if the substring with the current starting index (which begins as the n-1 for a string of length n) is valid. If so, we check if the complementary substring is a word (if it is, we have a valid hashtag and we return true). If substring with the current starting index was not a valid hashtag, we change its value in a boolean array of length n to false (it's set to true by default), so that we don't repeat this case. When checking if a recursive case is a valid hashtag, we check its value in the boolean array. Since there are a quadratic number of non-repeated cases, our runtime is quadratic.

Task 7.1



Task 8.1

```
(x + (x % 5) * 4) % 7

(84 + (4) * 4) % 7 = 100 % 7 = 2
(44 + (4) * 4) % 7 = 60 % 7 = 4
(66 + (1) * 4) % 7 = 70 % 7 = 0
(16 + (1) * 4) % 7 = 20 % 7 = 6
(105 +(0) * 4) % 7 = 105 % 7 = 0 (COLLISION)
(50 + (0) * 4) % 7 = 50 % 7 = 1 (COLLISION)
```

Index	Value
0	66
1	105
2	84
3	50
4	44
5	
6	16

Task 8.2

```
(x + (x % 5) * 4) % 7

(84 + (4) * 4) % 7 = 100 % 7 = 2
(44 + (4) * 4) % 7 = 60 % 7 = 4
(66 + (1) * 4) % 7 = 70 % 7 = 0
(16 + (1) * 4) % 7 = 20 % 7 = 6
(105 +(0) * 4) % 7 = 105 % 7 = 0 (COLLISION)
(50 + (0) * 4) % 7 = 50 % 7 = 1 (COLLISION)
```

Index	Value
0	66
1	105
2	84
3	
4	44
5	50
6	16

Task 9.1 Since each ID takes up 4 bytes, we have a total of 16 billion bytes needed to store all of these integers in memory, which is equivalent to 16 GB. Since I'm completing this assignment

on a computer with 8 GB of RAM, it would almost certainly crash.

Task 9.2 Since each 8 IDs take up 1 byte, we have a total of 0.5 billion bytes, or 500 MB, needed in total. This would certainly run on the computer I'm completing this assignment on, and it would probably run on my phone, which has 3 GB of RAM, as long as the phone isn't simultaneously running many other memory-intensive applications.

Task 9.3 At the start, the time complexity, both for allocation and release, is O(1), because release involves accessing and manipulating a single bit, and allocation will always find an ID in a constant (bounded above) number of operations. As the table approaches its capacity, release is still O(1), because it still must only access and manipulate a single bit, but allocation is O(n), because in the worst case, it might have to search through the entire table to access an unused ID. Changing the hash function would not help, because it is impossible for the hash function to know which integers are allocated at a given time, and in the worst case, it would still have to search through the entire table to find an ID for allocation.

Task 9.4 Let C denote the number of comparisons to find an ID when there are 5 free IDs left. We have that:

$$\begin{aligned}
\mathbb{E}[C] &= \sum_x \Pr(C = x)x \\
\mathbb{E}[C] &= \Pr(C = 1) * 1 + \Pr(C \neq 1) * (1 + \mathbb{E}(C)) \\
\mathbb{E}[C] &= (\frac{5}{M}) * 1 + \frac{M-5}{M} * (1 + \mathbb{E}(C)) \\
\frac{5}{M} * \mathbb{E}[C] &= 1 \\
\mathbb{E}[C] &= \frac{M}{5}
\end{aligned}$$

Hence, the expectation is $\frac{M}{5}$, or 800 million comparisons, to find an unallocated ID.

Task 9.5 To maintain a fast runtime and low memory consumption, one solution is to use a bitmap in conjunction with a queue. We maintain a queue of, say, 100 free IDs, as well as a bitmap array detailed earlier in Section 9, where each index corresponds to an ID, and the value of each bit (part of an 8-bit segment) is set to 1 if the ID is free.

Memory: In terms of memory consumption, storing 100 integers in queue consumes a trivial amount of memory: if each integer takes up 4 bytes, we are storing less than a kilobyte of data. The bitmap, as discussed in Task 9.2, only takes up 500 MB of memory. Therefore, our memory consumption is much better than that of a queue storing all free IDs.

Implementation: The implementation of release now changes. If the length of the queue is less than 100, we enqueue the ID passed to release. Otherwise, we access the index of the bitmap corresponding to the ID and change its value to 1. The implementation of alloc also changes. First, we check the length of the queue. If it is greater than 0, we call dequeue to remove the first element, and then we return that ID. Otherwise, we call a helper function replenishQueue, which performs a linear scan through the bitmap. Each time we find a free ID, we change its corresponding value

to 0 to indicate it has been taken, and then enqueue the ID. As soon as we find 100 IDs or reach the end of the array, replenishQueue returns. We then check the length of the queue. If it is still 0, we return -1 to indicate failure. Otherwise, we call dequeue to remove the first element and then return that ID.

Psuedocode: Since we haven't yet learned bit operations, I will assume we're dealing with an array of M bits for the bitmap in the pseudocode.

```
void release (id, queue, bitmap)
    if (queue -> length < 100)
        enqueue(queue, id);
    else
        bitmap[id] = 1;
    return;

int alloc (queue, bitmap)
    if (queue -> length > 0)
        return dequeue(queue);
    else
        replenishQueue(queue);
        if (queue -> length > 0)
            return dequeue(queue);
        else
            return -1;

void replenishQueue (queue, bitmap)
    int numFree = 0;
    for (int i = 0; i < M; i++)
        if (bitmap[i] == 1)
            bitmap[i] = 0;
            enqueue(queue, i);
            numFree++;
        if(numFree == 100)
            return;
    return;
```

Runtime: The runtime of release is clearly constant. It either enqueues an ID or changes a value in the bitmap. Since release constantly tops up the queue, replenishQueue, which runs in O(n) time, is called infrequently. When replenishQueue is not called, alloc runs in O(1) time, no matter the number of remaining IDs. If replenishQueue is called, the next 100 (or however many free IDs are left) calls to alloc are guaranteed to run practically instantaneously. Hence, the runtimes of both alloc and release are far better than those when using a bitmap, with both practically running in constant time.